Exploring Causes of Performance Overhead During Dynamic Binary Translation

Surya Tej Nimmakayala

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Committee:

Dr. Prasad Kulkarni: Chairperson

Dr. Bo Luo

Dr. Fengjun Li

Date Defended

The Thesis Committee for Surya Tej Nimmakayala certifies
That this is the approved version of the following thesis:

Exploring Causes of Performance Overhead During Dynamic Binary Translation

Committee:

Chairperson

Date Approved

Acknowledgements

I am thankful to my advisor Dr. Kulkarni for supporting me and guiding me throughout my research. I consider him as my mentor too, as he helped me to keep going at times even when the results were not as expected. He has been persistent in pointing my research compass in the right direction, not to be side-tracked with the numerous differentials we came across during the research.

I would also like to thank all the Professors under whom I have taken my courses, which helped in gaining exposure to the areas necessary to do my research. It has not only helped me with my research, but also gave me a better understanding of computer science as a whole and how the different fields have the potential in taking the existing technology to the next level.

Finally, would like to thank the students around for striving to work hard towards better things, to whom I could look upto at times when my morale was low.

Abstract

Dynamic Binary Translators (DBT) have applications ranging from program portability, instrumentation, optimizations, and improving software security. To achieve these goals and maintain control over the application's execution, DBTs translate and run the original source/guest programs in a sand-boxed environment. DBT systems apply several optimization techniques like code caching, trace creation, etc. to reduce the translation overhead and enhance program performance at run-time. However, even with these optimizations, DBTs typically impose a significant performance overhead, especially for short-running applications. This performance penalty has restricted the more wide-spread adoption of DBT technology, in spite of its obvious need.

The goal of this work is to determine the different factors that contribute to the performance penalty imposed by dynamic binary translators. In this thesis, we describe the experiments that we designed to achieve our goal and report our results and observations. We use a popular and sophisticated DBT, DynamoRio, for our test platform, and employ the industry-standard SPEC CPU2006 benchmarks to capture run-time statistics. Our experiments find that DynamoRio executes a large number of additional instructions when compared to the native application execution. We further measure that this increase in the number of executed instructions is caused by the DBT frequently exiting the code cache to perform various management tasks at run-time, including code translation, indirect branch resolution and trace formation. We also find that the performance loss experienced by the DBT is directly proportional to the number of code cache exits. We will discuss the details on the experiments, results, observations, and analysis in this work.

Contents

List of Figures

Chapter 1

Introduction

The Dynamic Binary Translation (DBT) systems provide a *sandboxed* environment for binary program execution. Sandboxing in DBTs is a mechanism to control and monitor program execution. To achieve control over program execution, a DBT interjects normal execution, translates and/or instruments the original (*guest*) binary code, and runs this cached translated binary copy instead. DBT systems have found numerous uses in program instrumentation and profiling [6, 24], program optimization [2], binary portability [3, 28, 31] and secure execution [21, 23].

Dynamic binary translators need to perform several other tasks, in addition to executing the guest program. These additional tasks include the just-in-time (JIT) translation of the guest code to the host format and the resolution of the direct and indirect control-flow transfers to use the new translated addresses instead. These tasks impose overhead at run-time and slow-down the execution of the translated program. Any instrumentation added to the translated code to perform profiling or security operations further exacerbates the performance issue for DBTs. Poor performance restricts the applicability and attractiveness of DBT systems in spite of their potential benefits.

Researchers have developed several techniques and optimizations to improve DBT performance. These techniques include the use of a software code cache to store previously translated code blocks [26], chaining direct branches in the code cache [7], mechanisms to predict indirect branch targets [19], and combining multiple code cache blocks into *traces* to improve cache locality [2]. While these approaches have resulted in dramatically improving DBT performance, it still significantly trails the performance delivered by native code execution for many programs, especially those that are short-running [6, 29].Therefore, there is still a need to explore and develop new techniques to improve DBT performance and allow its broader and mainstream adoption.

The goal of this work is to study the inner workings of state-of-the-art dynamic binary translators to understand the primary reasons contributing to their performance bottlenecks. Similar studies were conducted in the past evaluate the benefits of individual optimization techniques in a DBT [6, 18]. Other researchers have also attempted to understand the impact of a DBT on microarchitectural structures, like instruction caches and translation lookaside buffers [26]. In this thesis, in addition to validating these past studies on latest DBTs and machine architectures, we also perform a new instruction-level analysis of how and where a DBT spends its time. We detect and show the primary contributors to the performance loss seen by the DBTs, and correlate such loss to its contributing factors. We expect that the knowledge gained from our study will enable researchers to develop new strategies to overcome DBT performance bottlenecks on modern machines.

We conduct our experiments using a popular and sophisticated x86 to x86 dynamic binary translator, called DynamoRio [6]. DynamoRio has been heavily optimized and implements most of the main stream techniques to improve DBT performance. In addition, we employ standard tools, such as `perf` and `ptrace`, available on most Linux distributions to perform our tests.

The rest of this thesis is organized as follows. We describe our experimental setup and benchmark information in the next section. We then present our experimental results, and discuss observations and findings in Chapter 3. We present other related works in Chapter 4. Finally, we describe the future work and our conclusions from this study in Chapters 5 and 6 respectively.

Chapter 2

Framework for Capturing Statistics on Dynamic Binary Translation

In this chapter we describe our experimental framework, including the tools we use and the benchmarks we employ. We provide an introduction to the design and implementation details of our selected dynamic binary translator (DBT). We also explain the measurement tools, along with their important configuration parameters in this chapter. In order to explore the impact of dynamic binary translation, we use the open-source DynamoRIO framework. We capture different statistics to find the effective slow-down caused by DynamoRIO for our benchmark programs, and the impact on the hardware, to help determine the possible causes of execution overhead. We expect these causes to give us the basis to make further enhancements to the DynamoRIO tool to improve its performance.

2.1 DynamoRIO design details

One basic goal of *Dynamic Binary Translation* is to overcome the performance drawback of interpretation with the help of caching and optimizing the translated instructions. This is especially helpful for longer running programs, where translation can avoid significant interpretation cost during program execution by caching the translated code in a software cache. However, dynamic translation involves an overhead as extra work has to be done in translating and caching the *translation blocks/instructions*. Depending on the number of translations performed by the DBT, the quality of the translated code generated, and the total run-time of the application, the overall execution can either see a slow-down or a speed-up as compared to native program execution outside the DBT.

We have chosen DynamoRIO [12] as the platform to perform the required *Dynamic Binary Translation* and study its impact on the program execution and determine the probable reasons for the overhead. The tool can be used for a variety of applications like profiling, optimization and security. DynamoRIO offers an API [16], that can be used for the necessary code instrumentation. We can write *clients* [11] depending on our application of the tool, which provide instrumentation for various events that can occur during program execution. We will describe the client interface further in section 2.2.3.

When an application runs under DynamoRIO, the tool handles the application code as fragments. There are two types of fragments that DynamoRIO creates during the application execution. One is the *basic block* and the other is the *trace*. An example *basic block* is shown below:

```
TAG  0x00007f17534e06f0
 +0    m4 @0x0000000053ca1a50  ff 05 b2 36 56 1e    inc    <rel> 0x00000000722020b8 -> <rel> 0x00000000722020b8
 +6    L3                    31 c0                xor    %eax %eax -> %eax
 +8    L3                    48 8b 5c 24 28       mov    0x28(%rsp) -> %rbx
 +13   L3                    48 8b 6c 24 30       mov    0x30(%rsp) -> %rbp
 +18   L3                    48 83 c4 38          add    $0x0000000000000038 %rsp -> %rsp
 +22   L3                    c3                   ret    %rsp (%rsp) -> %rsp
END 0x00007f17534e06f0
```

The instructions marked as *L3* are the original application instructions and the first instruction marked as *m4* is instrumentation injected by the client to increment a counter. As can be seen, a basic block is a block of instructions that execute in sequence and end when there is a change in the flow with a jump to a different part of the code. A trace constitutes one or more basic blocks that signify hot code, or code executed more frequently during the program execution. These fragments after being translated are stored in their respective code caches. So, we have two types of caches: *Basic Block Cache* and *Trace Cache*. This would prove advantageous the next time a stored fragment is executed, as it does not have to be translated again and can be directly executed from the respective code cache.

As shown in the example basic block, each fragment has a *TAG*. This *TAG* is the source binary address (SPC) of the first instruction in the block, and keeps track of the application code address to be reached next for execution. Since the application can not be executed directly and needs to be executed from within DynamoRIO, the *TAG* of the fragment helps the tool know if a fragment for the next application instructions to be executed is already present in the code cache and accordingly executes or creates a new fragment in case there is no entry for the *TAG*. The mappings between the *TAG* of the fragments and the translated block address (TPC) in the code cache are maintained in lookup tables.

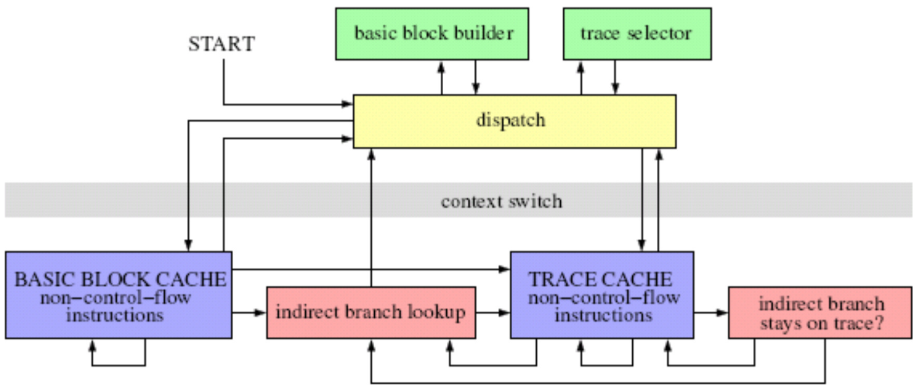

Figure 2.1. DynamoRIO execution flow from [17]

Figure 2.1 from [17] shows the execution flow of DynamoRIO. As a program executes under DynamoRIO, the *dispatch* part of the code gets the first/next application code address (SPC) to be executed. *dispatch* performs a lookup to check if a fragment already exists in the code cache for the application address provided as TAG. If there is an entry, a ćontext switchís performed from DynamoRIO code to the corresponding fragment code cache to execute the translated code. If there is no entry for the application address code, translation is done to create a new fragment and the lookup tables and code cache are updated with the new entry. A context switch then takes the execution to the first intruction of newly created fragment. In the *dispatch* routine a check for *trace selection* is also done to see if the code in the frgament is hot enough to qualify as a trace head. If it is hot enough, a new trace is created with the corresponding basic block as the *trace head*. The consecutive basic blocks executed are added to the new trace until a trace termination condition is met. DynamoRIO considers a basic block executed for 50 times as hot code to start a trace.

A significant source of overhead in DynamoRIO results from the excessive transitions between the DynamoRIO emulation engine and the code cache for a variety of reasons other than actual code translation. This means that any added advantage of the caching of translated fragments can be nullified by this additional execution of DynamoRIO code that does not actually contribute to the dynamic binary translation and is a side-effect of the same. Probably with this in mind, the developers of DynamoRIO have made attempts to keep the control within the code cache for most of the execution time by linking fragments in the code cache in case of direct jumps and also instruction inlining for indirect branches by resolving the same through appropriate checks and maintaining information on the indirect branches in lookup tables stored within the code cache. Inspite of these attempts we do see significant overhead with a number of benchmarks run under DynamoRIO.

2.2 DynamoRIO changes for experiments

We have run experiments to collect data and have analyzed the same to know the possible causes of overhead during a DynamoRIO run. We will discuss the results of these experiments and our observations in Chapter 3. To perform our experiments we had to make some changes to the DynamoRIO source code, including the code of an existing DynamoRIO client and also used the Linux *PERF* [25] tool to find the influence on hardware micro-architectutral features and corresponding performance impact. All the runs of benchmarks under DynamoRIO have been done through a client, except for one experiment where the environment variables have been used to do the required set-up for the benchmark runs under DynamoRIO. We discuss the code changes and the set-up used for each of the experiments in the following sections.

2.2.1 Run times and Compilation times

The first experiment conducted was to capture the execution times of the benchmarks run natively (outside DynamoRIO) and run under DynamoRIO with and without trace formation enabled. For this,we have used the *time* [8] command with the three different runs. To turn-off the traces during execution from within DynamoRIO the run-time option *disable traces* [14] had been used. Sample commands for the three runs are given below:

```
Native Run: time ./perlbench -I. -I./lib attrs.pl
DynamoRIO run with traces: time drrun perlbench -I. -I./lib attrs.pl
DynamoRIO run without traces: time drrun -disable_traces -- perlbench -I. -I./lib attrs.pl
```

The above commands are to run the *400.perlbench* benchmark for the test input *attrs.pl* in three different configurations. The application is run through DynamoRIO with the help of *drrun* [10] which is a tool of DynamoRIO. It sets up the required configuration and runs the application from within DynamoRIO.

We added some code to the *dispatch* function in the DynamoRIO source file **core/dispatch.c** to capture the compilation time. The code involves use of the *gettimeofday* function to get the time difference between the start and end of each basic block translation and print out the cumulative time of translation for all the basic blocks during execution.

2.2.2 Perf statistics

We have used the Linux *perf* tool to capture the statistics of different hardware counters during the execution of benchmarks under DynamoRIO and the native run. The hardware counters would be captured for different events specified along with the *perf* command during benchmark execution. However, the number of events that can be given in one execution, without any skewing of the hardware counters depends on the number of hardware counters in the system. We could run up to 5 events on our system in one execution of the *perf* command. So, in order to capture the data on all the desired events, we had to make 3 sets of events and run each benchmark 3 times with the *perf* command.

Below are the sample *perf* commands for native execution of *400.perlbench* benchmark, test input *attrs.pl*:

```
perf stat -B -e cycles,instructions,branches,branch-misses perlbench -I. -I./lib attrs.pl
perf stat -B -e cycles,instructions,cache-references,cache-misses perlbench -I. -I./lib attrs.pl
perf stat -B -e page-faults,context-switches,L1-icache-load-misses,iTLB-load-misses,
cache-misses perlbench -I. -I./lib attrs.pl
```

2.2.3 Application Instruction Counts

We used the client binary *libbbcount.so* along with the **drrun** tool to capture the number of application instructions executed from within DynamoRIO. This is an existing client given with the standard DynamoRIO source code distribution. We made a few changes to this client code to capture the application instruction counts.

A client is an interface through which certain events [13] within DynamoRIO can be registered for the particular execution of an application. Example of common events include *basic block creation event*, *trace creation event* etc. When we register an event through the client, DynamoRIO performs a call-back to the registered function when the event occurs at run-time. So, these call-back functions will have the code to perform any desired actions for that event. The DynamoRIO

API can be used to perform any action from within these call-back functions. We have also added some additional functions to the API to be called from within the client.

We have used the DynamoRIO client to register three events: program exit, basic block creation and trace creation. There are seperate functions registered for each of these events done in the **dr_init** function within the client. The client source file is **api/samples/bbcount.c** and below are the functions registered for the three events:

```
dr_register_exit_event(event_exit);
dr_register_bb_event(event_basic_block);
dr_register_trace_event(event_trace);
```

The arguments to the above functions are the call-back functions that DynamoRIO calls on occurrence of these events at run-time. We have used the call-back function of basic block creation to inject code into the basic blocks to keep count of application instructions executed from within the code cache. With the exit event call-back function, the final count is being printed out.

Below are the sample commands used to run the *400.perlbench* benchmark test input *attrs.pl*, from within DynamoRIO through the client binary *libbbcount.so*.

```
Native Run: time ./perlbench -I. -I./lib attrs.pl
DynamoRIO run with traces: time drrun -client libbbcount.so 0 "" perlbench -I. -I./lib attrs.pl
DynamoRIO run without traces: time drrun -client libbbcount.so 0 "" -disable_traces -- perlbench
-I. -I./lib attrs.pl
```

2.2.4 Fcache exit data from logs

We have extracted data from the DynamoRIO logs on code cache exits. This data helped us to know the major reasons why the control leaves the code cache during application execution from within DynamoRIO. To capture the DynamoRIO logs the option *loglevel 3* has been used.

2.2.5 Code cache instructions with ptrace

2.2.6 Code cache instructions with ptrace

We have used *ptrace* [15] to monitor the execution of benchmarks under DynamoRIO. The purpose of this experiment was to capture the information on the number of instructions executed for different kinds of exits from the DynamoRIO code cache. In order to mark the different entry points in DynamoRIO after the exit from code cache, dummy function's have been added to the DynamoRIO code at appropriate places. Then, using the command *nm -a -v* [1] on the DynamoRIO shared library we could find the memory addresses for the dummy functions. The memory addresses are essential for the parent process to monitor the content of *rip* (instruction pointer) register during the execution of child process, and thereby mark the different phases of execution that correspond with the code cache exit type. Once we have marked the different phases, corresponding instruction counts can be captured with an increment operation for each instruction executed.

We built a separate script to use the *ptrace* Linux tool for monitoring, where a child process is forked [30] and the environment for benchmark run through DynamoRIO is set-up. In the child process, the *ptrace* command with the request *PTRACE_TRACEME* is executed, enabling the parent to monitor the benchmark execution through DynamoRIO. The content of *rip* register is gathered with the *ptrace* command excuted with the request *PTRACE_GETREGS*. The */proc/child pid/maps* file is used to determine the information of the corresponding segment being reached during execution by comparing the start and end addresses in the file with the contents of *rip* resgister. Similarly, the different phases have been marked by comparing the dummy function addresses (captured with *nm* command) with the address in *rip* register. After the execution phase has been marked, the parent sends the *ptrace* command with request *PTRACE_SINGLESTEP* to the child process, so that for each instruction executed a counter can be incremented which would essentially give us the executed instruction count in a particular phase.

While executing the benchmarks with their smaller test inputs, it is possible to single-step the entire execution to capture the exhaustive instruction count information, as the test inputs produce short-running program runs. However, when running programs with the reference inputs, single-stepping through the entire execution is not feasible as they produce long-running programs and might take weeks to complete the execution. So, for the reference inputs, the single-stepping has been done in intervals of 100 million instructions. After every 100 million instructions, the parent sends *ptrace* command with request *PTRACE_CONT*, in order for the child process to continue normal execution, which is the fast-mode execution (compared to the single-step execution). During the fast-mode execution, parent process sleeps [33] for that duration. Once the fast-mode time is completed, parent sends the signal *SIGTRAP* to the child, through the *kill* [32] command. This way, the parent process will regain control over the child process and continue to monitor its execution.

Another aspect to consider is the fast-mode time for different benchmark reference inputs. As the execution times for different benchmark-input pairs are different, to get enough samples of data, the fast-mode times have been computed to record data for about 50 intervals of 100 million instructions each during every program execution. The monitoring process sleeps for the calculated period between successive intervals. The information on the instruction counts for different DynamoRIO phases (inline with the code cache exits) is printed out for every 0.5 million instructions with test inputs and 100 million for the reference inputs.

Chapter 3

Experimental Results and Analysis

To gather the required scientific data, we have added custom code to the original DynamoRIO source code. This custom version of DynamoRIO code coupled with a DynamoRIO client is run on each of the benchmarks to collect the data on program run-times, hardware counters (perf), and application instruction counts. For the experiment to collect the instruction counts executed in different memory segments using *ptrace*, instead of the client, environment variables have been used to set-up the run of the benchmark under the control of DynamoRIO.

We present our analysis by compiling different sets of data to harvest the different factors that possibly lead to the run-time overhead (or speed-up) of different benchmarks run through DynamoRIO.

3.1 Experimental Setup

Our experimental platform consists of a cluster of Intel Xeon (R) W3530 2.8 GHz (x 8 cores) work-station's with 3.9 GB memory, running the 64-bit Fedora 18 operating system. We use the industry-standard SPEC integer benchmarks to collect our performance results [9]. Each benchmark is provided with two sets of inputs, *test* and *ref.* Test inputs typically provide short program run-times, while 'ref' inputs are used for long-lived program runs.

3.2 x86 Results and Analysis

In this section we discuss and analyse the collected data represented as graphs, pertaining to different factors contributing to the run-time overhead.

3.2.1 Benchmark Statistics - Test Inputs

Figure 3.1 shows the ratio of DynamoRIO run-times to Native run-times for different benchmarks run with test inputs. This basically gives an insight to the overhead of running the program through DynamoRIO. If the ratio is less than 1, it signifies a speed-up and a ratio greater than 1 signifies an overhead.

Test inputs are short running programs that account for the case of start-up overhead when running different programs through DynamoRIO. The benchmarks with substantial overhead are the 400.perlbench, 403.gcc and 445.gobmk. While the highest overhead is for the 400.perlbench input regmesg.pl (with traces), we could even see a speedup with the 445.gobmk input capture.tst. In this section, we discuss our analysis on the factors responsible for the overhead. We also suggest possible ways to reduce the overhead, or in other words to try and get the execution state of benchmarks with overhead to the one with the speedup.

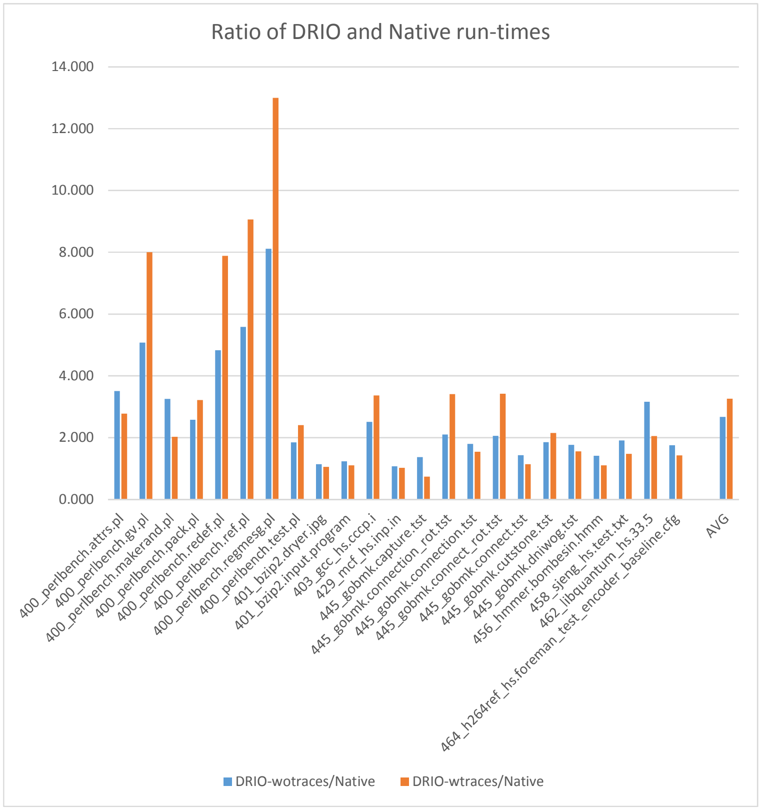

Figure 3.1. Ratio of *DynamoRIO* run times to *Native* run times for different benchmark test inputs

The graph in figure 3.2 plots the compilation times for different benchmarks to see if it is a major contributor for the performance overhead. It can be seen that the time taken for compilation or translation of the source binary to target binary is indeed the major contributor to the overhead with the test inputs. Target binary is essentially basic blocks of source binary instrumented with additional instructions, in order for DynamoRIO to maintain control over the execution flow of the source binary. DynamoRIO incorporates the concepts of tracing and linking to the target binary at the fragment level (fundamental unit of target binary).

From the Figure 3.2, it is seen that for benchmarks that have the highest

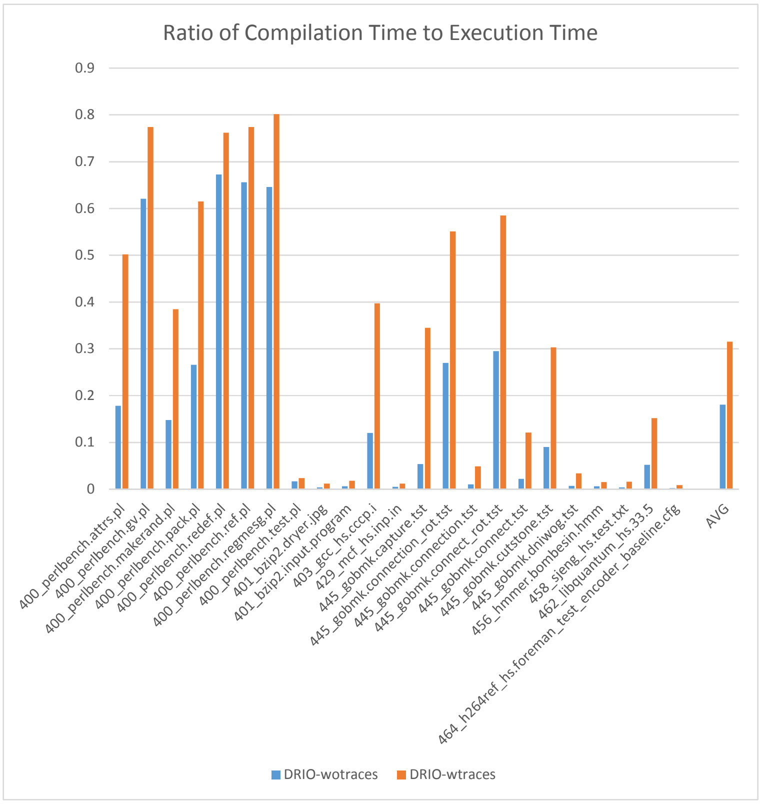

Figure 3.2. Ratio of *Compilation* time to *Execution* time for different benchmark test inputs

percentage of overhead, like 400.perlbench with test input regmesg.pl, has the highest ratio of compilation time which is 80% of execution time. Similarly, the other test inputs for benchmarks 400.perlbench, 403.gcc and 445.gobmk also have significant compilation times. This high percentage can also account for the size of their executables, which are 1203 KB, 3632 KB and 3944 KB respectively. The size of executables for each of the benchmark can be seen in Table 3.1, and also the respective native run times.

Figure 3.2 also illustrates that the tracing mechanism incorporated does not result in optimized performance but rather adds overhead, as the compilation times are considerably high when run with traces than without traces.

As these are short running programs, the overhead of compilation can not be

Benchmark	Static Exec. Size(in KB)	Trace Count	Basic Block Count	Native Run-Time(sec)	Total Context Switches
400_perlbench.attrs.pl	1203.066	1492.700	16627.100	0.147	151708.800
400_perlbench.gv.pl	1203.066	1008.700	13779.400	0.025	105540.600
400_perlbench.makerand.pl	1203.066	263.300	5855.200	0.077	33504.200
400_perlbench.pack.pl	1203.066	3666.200	26878.400	0.193	291185.000
400_perlbench.redef.pl	1203.066	528.500	9307.800	0.017	69650.500
400_perlbench.ref.pl	1203.066	707.000	10717.800	0.017	78645.500
400_perlbench.regmesg.pl	1203.066	1102.400	15261.800	0.017	119580.100
400_perlbench.test.pl	1203.066	1026.755	13262.705	4.399	104746.373
401_bzip2.dryer.jpg	72.381	554.500	3567.500	4.104	38202.200
401_bzip2.input.program	72.381	516.100	3339.700	2.466	37635.500
403_gcc_hs.cccp.i	3632.383	20964.200	103816.200	1.354	1385432.600
429_mcf_hs.inp.in	22.643	248.000	2569.000	2.795	19392.000
445_gobmk.capture.tst	3944.371	1631.100	8778.200	0.548	111445.000
445_gobmk.connection_rot.tst	3944.371	1593.000	9098.900	0.071	106304.400
445_gobmk.connection.tst	3944.371	4198.800	19469.000	4.634	272542.600
445_gobmk.connect rot.tst	3944.371	1698.000	9322.100	0.069	112671.700
445_gobmk.connect.tst	3944.371	2671.800	12343.500	1.647	169207.300
445_gobmk.cutstone.tst	3944.371	2215.000	10623.500	0.269	143610.500
445_gobmk.dniwog.tst	3944.371	7055.000	31364.200	11.859	452076.900
456_hmmer.bombesin.hmm	314.469	309.000	4202.800	2.927	29803.400
458_sjeng_hs.test.txt	149.578	1108.000	5782.200	4.065	72226.000
462_libquantum_hs.33.5	50.480	205.000	2119.200	0.082	17404.000
464_h264ref_hs.foreman_test.cfg	565.682	1960.000	11694.200	15.406	149200.000

Table 3.1. Different stats related to the benchmark test inputs

compensated with the resulting improved code locality produced by DynamoRIO. The time taken for compilation is more than the actual execution time for some benchmark/inputs resulting in the high overheads for most of the benchmarks. Furthermore, the high compilation times is the result of high context switches between the contexts of DynamoRIO and the code cache (application code execution). This high number of context switches in turn results in the execution of DynamoRIO code for most of the execution time. Also, the percentage of context switches to actually create the fragments (basic blocks, traces) is less.

The table 3.1 has information on the number of traces, and basic blocks created for each of the benchmark, as well as the number of context switches between the context of DynamoRIO and application execution from the translated code in caches. It can be seen that the context switches are significantly more than the sum of the traces and basic blocks, supporting our earlier statement on the context switches.

We researched further to find the possible causes of context switches other than the actual compilation. The log data collected from DynamoRIO on the code cache exits has been helpful to know the major reasons for control exiting code cache for reasons other than the fragment compilation.

Figure 3.3 shows the ditribution of different factors causing the control exit from the fcache. The figure also shows the average of all the factors for all the benchmarks, showing the important factors for fcache exits.

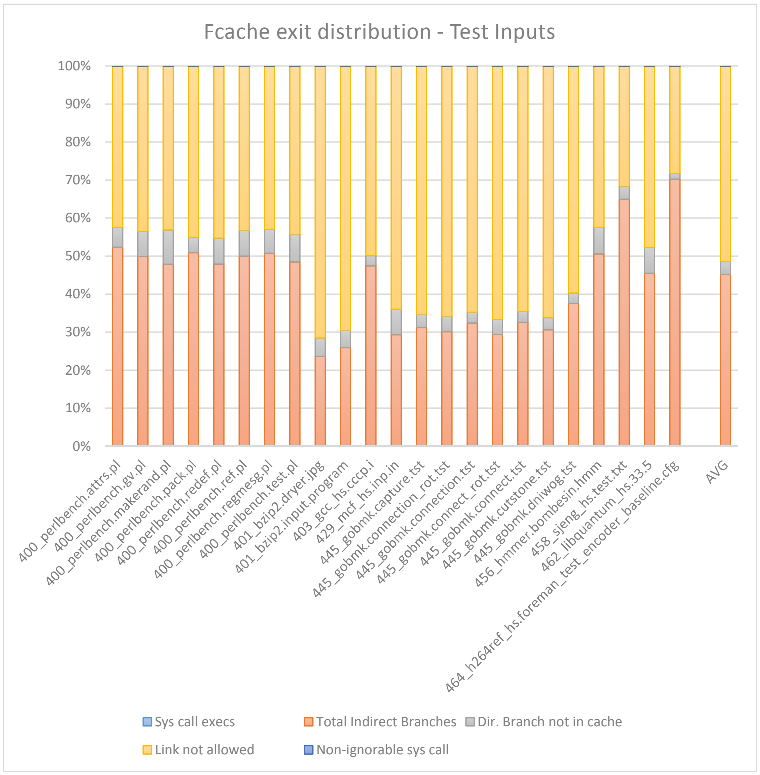

Figure 3.3. Distribution of fcache exit causes for different benchmark test inputs

It can be seen that there are two outstanding factors, *Indirect Branches and Link not allowed between fragments* in addition to block translation. Intuitively, the number of additional instructions executed should depend on and correlate with the number of code cache exits. But, we could see that even for some benchmarks with similarly high exit rates, the additional instructions executed by DynamoRIO are signficantly different.

Benchmark	Total Fcache Exits/millisec	perf instrcnt ratio(DRIO/Native)
400_perlbench.attrs.pl	379.75	73.89
400_perlbench.gv.pl	536.48	89.36
400_perlbench.makerand.pl	213.91	3.60
400_perlbench.pack.pl	484.77	10.37
400_perlbench.redef.pl	523.49	167.49
400_perlbench.ref.pl	514.41	130.47
400_perlbench.regmesg.pl	556.86	105.95
400_perlbench.test.pl	5.46	16.28
401_bzip2.dryer.jpg	8.64	1.27
401_bzip2.input.program	13.52	1.31
403_gcc_hs.cccp.i	309.66	3.73
429_mcf_hs.inp.in	6.71	1.52
445_gobmk.capture.tst	279.76	3.24
445_gobmk.connection_rot.tst	428.81	5.76
445_gobmk.connection.tst	38.50	1.60
445_gobmk.connect rot.tst	471.23	6.78
445_gobmk.connect.tst	91.46	1.70
445_gobmk.cutstone.tst	248.40	2.65
445_gobmk.dniwog.tst	24.36	1.52
456_hmmer.bombesin.hmm	9.48	1.28
458_sjeng_hs.test.txt	12.63	1.70
462_libquantum_hs.33.5	95.62	1.82
464_h264ref_hs.foreman_test.cfg	14.73	1.53

Table 3.2. *Fcache exit ratio* against *perf instruction count ratios* for different benchmark test inputs

Table 3.2 shows the exit rates for different benchmark test inputs along with the corresponding *perf* instruction count ratio between DynamoRIO to Native instruction counts. It can be seen from this table that for the benchmark *400.perlbench* test input *attrs.pl* the exit rate is *379.75* and for the benchmark *403.gcc* test input *cccp.i* the exit rate is *309.66.* Though the exit rates are close, there is a big difference between the *perf* instruction count ratios.

Table 3.3 shows the ratio of DynamoRIO to Application *perf* counters for different events. It can be seen that for the benchmark *400.perlbench* test input *attrs.pl* the **L1 icache load misses and iTLB load misses** are more compared to the benchmark *403.gcc* test input *cccp.i.* This probably explains the big difference in the *perf* cycle count ratios as the high cycle count for *400.perlbench* can be due to the stall cycles required to address the penalty of the misses. The misses also show that the code had been accessed more randomly compared to the other benchmark. The overhead is probably not indicated by just the exit rate and the outstanding factors of the exits, but also on how the hardware is able to handle the extra load of instructions considering the resultant code locality by running the program from within DynamoRIO.

Benchmark	Branch Miss%	Cache Miss%	Context Switches	L1 icache load misses	iTLB load misses
400_perlbench.attrs.pl	0.543	0.371	5.743	78.396	65.216
400_perlbench.gv.pl	0.663	0.704	9.325	109.491	83.534
400_perlbench.makerand.pl	0.514	0.450	16.750	201.198	21.938
400_perlbench.pack.pl	0.819	0.186	3	17.029	48.106
400_perlbench.redef.pl	0.551	0.511	33.600	178.569	75.824
400_perlbench.ref.pl	0.618	0.341	33.400	152.658	101.630
400_perlbench.regmesg.pl	0.598	0.563	12.100	123.410	59.279
400_perlbench.test.pl	1.134	0.668	1.092	28.083	19.136
401_bzip2.dryer.jpg	0.864	1.295	4.768	21.004	5.378
401_bzip2.input.program	1.001	1.390	5.207	23.617	4.529
403_gcc_hs.cccp.i	0.988	0.659	6.804	15.448	28.197
429_mcf_hs.inp.in	0.709	1.001	3.814	12.851	3.097
445_gobmk.capture.tst	0.928	0.178	2.720	32.802	38.289
445_gobmk.connection_rot.tst	0.516	0.130	15.727	51.631	69.832
445_gobmk.connection.tst	1.014	0.090	1.532	11.033	11.371
445_gobmk.connect rot.tst	0.684	0.125	11.581	67.363	89.123
445_gobmk.connect.tst	0.918	0.150	1.419	12.917	14.543
445_gobmk.cutstone.tst	0.715	0.119	3.908	16.143	33.563
445_gobmk.dniwog.tst	0.983	0.375	2.608	9.500	7.873
456_hmmer.bombesin.hmm	0.826	1.309	3.219	17.508	2.455
458_sjeng_hs.test.txt	0.886	0.569	5.661	43.097	5.974
462_libquantum_hs.33.5	2.612	0.403	33.300	183.327	35.996
464_h264ref_hs.foreman_test.cfg	1.026	1.033	2.728	7.166	3.319

Table 3.3. Ratio of *DynamoRIO* and *Application PERF* counters for different events with each of the benchmark test inputs

Table 3.4 shows the counts of other instructions executed with in the code cache apart from the translated instructions. The counts have been captured with the help of ptrace and maps. The ratio of other instructions to the total instructions is not that high, implying that the overhead due to the other instructions is not that big. The inputs for the benchmarks 401.bzip2 and 462.libquantum have the highest overhead with 11% and 10% respectively. Other inputs have the overheads in the single digits or less than 1.

Benchmarks	Code Cache - Other ICount	Total ICount	CC-Other ICount/Total ICount
400_perlbench.redef.pl	8983533.000	887665902.000	0.010
401_bzip2.input.program	1201238475.000	10855705330.000	0.111
403_gcc_hs.cccp.i	9646369.000	1024677672.000	0.009
429_mcf_hs.inp.in	123415.000	64442423.000	0.002
445_gobmk.connect rot.tst	3869729.000	251377454.000	0.015
445_gobmk.cutstone.tst	751636.000	233894825.000	0.003
456_hmmer.bombesin.hmm	223028.000	89912289.000	0.002
458_sjeng_hs.test.txt	17582478.000	336561573.000	0.052
462_libquantum_hs.33.5	48002097.000	476153483.000	0.101
464.foreman_test.cfg	144047.000	73680140.000	0.002

Table 3.4. Ratio of *Other code cache instructions* and *Total Instructions* of the benchmark test inputs

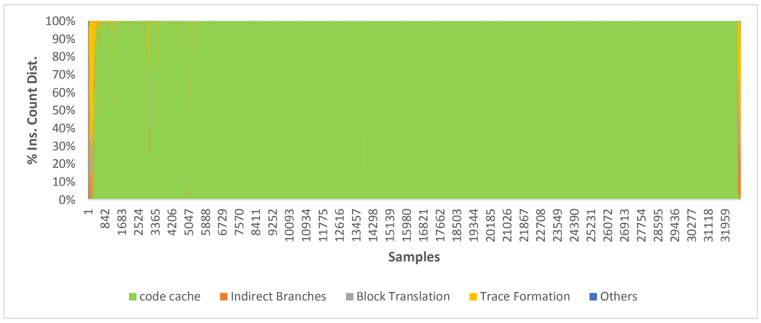

Figure 3.4. Distribution of instruction counts for different execution phases of 456.hmmer benchmark input bombesin

A better insight of how the different exits from the code cache influenced the benchmark execution would be through the number of DynamoRIO instructions executed with each of those exits. We have used *ptrace with single-step* to execute the benchmark and collect samples of instruction counts executed in different phases for every *0.5 million instructions*. With the use of *maps* that helped in keeping track of the memory segments along with the program counter enabled us in capturing the counts for different phases. The figure 3.4 shows the distribution of instructions counts for different execution phases of benchmark *456.hmmer* with input *bombesin*. The color code of different execution phases maps to the fraction of DynamoRIO instruction counts executed to address the corresponding fcache exit reasons: *Indirect Branchs,Block Translation and Trace Formation*. It can be seen that most of the graph is *green*, which is the color code for instructions executed in the *code cache*. This is a good indicator of high percentage of code being executed from within the code cache. The next significant phase executed is *Trace Formation* which relates to the fcache exit due to absence of linking between certain fragments and the trace heads to be executed next, followed by the phase for *Indirect Branches*.

Ideally, this distribution of instruction counts for different execution phases of DynamoRIO should be inline with the distribution of different fcache exits for the *bombesin* input of benchmark *456.hmmer* shown in figure 3.3. But, according to the fcache exit distribution, the instructions executed for exit due to *Indirect Branches* which is around 50% of the total exits, should be more compared to the exit due to *Trace Formation*, which is around 42% of the total exits. This contradicts with our findings on the instruction count distribution. The reason for this can be that the optimization technique of instruction inlining might be coming in handy, resolving the indirect branches within the fragment cache, averting the need to exit from the code cache, assuming that the design of the code is more structured enabling the indirect branch prediction to be more efficient. From this, it can be said that the number of DynamoRIO instructions needed to execute in resolving the reason for exit is not a constant for all the cases of a particular exit type, but also depends on the nature of the exit and other factors like the employed optimizations. So, the instruction count distribution is not directly proportional to the fcache exit distribution.

It is also possible to explain the low overhead with this particular benchmark run, resulting in a run-time close to that of native run from the figure 3.1. The optimization techniques like, caching of basic blocks, linking of basic blocks within cache, trace creation, instruction inlining to handle indirect branches from within the fragment cache must have helped in keeping most of the execution within the fragment cache and abating the effect of additional instructions executed apart from the application code. The design of the benchmark code also plays a role, as it is possible that the code of 456.hmmer is more structured and the test input invokes only part of the benchmark code base that is more re-usable and requires less translation.

Another interesting case to look at would be the benchmark *400.perlbench* run with test input *regmesg.pl*. This has the highest run-time overhead as shown in the figure 3.1 and also the major contributing factor for overhead is the compilation time or the basic block translation time, shown in figure 3.2. The fcache exit distribution for the same can be seen in the figure 3.3, which is around *52%* for *indirect branches*, *43%* for *trace formation* and *15%* for *block translation*.

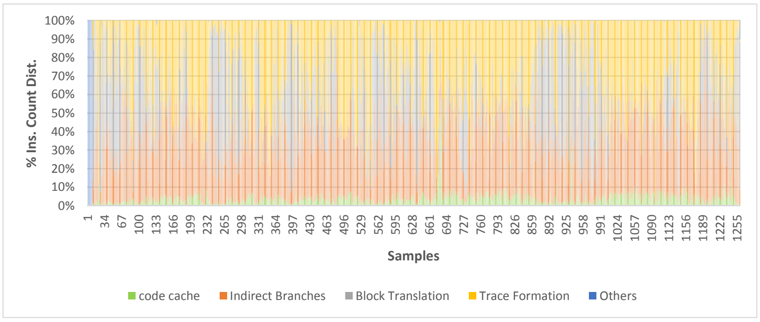

Figure 3.5. Distribution of instruction counts for different execution phases of 400.perlbench benchmark input regmesg.pl

Now, referring to the instruction count distribution during the benchmark execution in figure 3.5, we can see that the code executed during *block translation* is pervasive. This concurs with the high compilation time overhead mentioned earlier with reference to the figure 3.2. The other phases *Indirect Branches*, *Trace Formation*(color codes *orange* and *yellow* respectively) can be prominently seen in the distribution posing a high degree of overhead. As a result, very small portion of instructions have been executed from within the fragment cache which is the distribution in *green*.

Next, we discuss the affect of the high instruction volume on the hardware. Table 3.3 shows the ratio of branch misses with DynamoRIO compared to the native run. It can be seen from the table that the ratios for the benchmarks 400.perlbench, 403.gobmk and 445.gobmk are less than 1, indicating that the branch misses for native run are more compared to the run with DynamoRIO.

Similarly, the Table 3.3 has the cache miss ratios for different benchmarks. These ratios for the benchmarks 400.perlbench, 403.gobmk and 445.gobmk is less than 1, being coherent with the data of the branch miss percentages. It can be said that the improved code locality through the DynamoRIO run has resulted in an overall improvement of the branch prediction and also the overall cache misses for these benchmark test inputs.

However, other sources of penalty with respect to hardware are significant resulting in the overhead not being compensated. Table 3.3 shows the context switches for different benchmark test inputs run through DynamoRIO as a multiple of the context switches run natively. The context switches are more compared to native run. Similarly, the Table 3.3 gives information on the *L1 icache load misses* and *iTLB load misses* with the DynamoRIO runs against the native runs. It is probably because of the large executable size of the benchmark in comparision with the DynamoRIO code, resulting in a high percentage of L1 and TLB misses. The resulting penalty probably couldn't be compensated by any added speed with some other factors during execution like overall branch misses and overall cache misses causing the overhead with each of the benchmarks. Looking into remodelling the hardware can be one of the future explorations to ameliorate the execution time of the programs through DynamoRIO as the code base of the tool itself is large and can probably get even bigger with any new features. Also, exploring opportunities to parallelize parts of the DynamoRIO code can be beneficial.

3.2.2 Benchmark Statistics - Ref Inputs

The below figure 3.6 shows the ratio of DynamoRIO run times to the native run times. It can be seen that similar to the test inputs the highest overhead is with the benchmarks 400.perlbench, 403.gcc and 445.gobmk. However, the highest overhead is with the input diffmail.pl of the benchmark 400.perlbench, with close to 100% overhead.

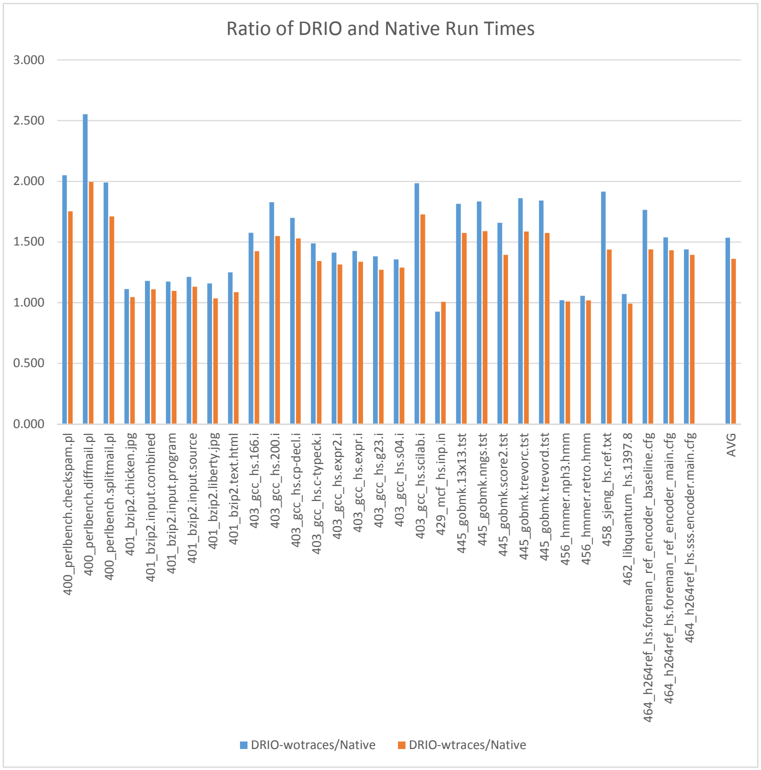

Figure 3.6. Ratio of *DynamoRIO* run times to *Native* run times for different benchmark ref inputs

Reference inputs signify the steady state run of a program and lead to long running times. So, the compilation overhead would be compensated by the longer execution times for each of the benchmarks. This aspect is illustrated in Figure 3.7, with ratios of the compilation times to the execution times for different benchmark reference inputs. The highest compilation time ratio's are for the

403.gcc benchmark with the input *scilab.i*, which is *0.071* or *7%*. This shows that the compilation time is not a significant contributor to the overhead.

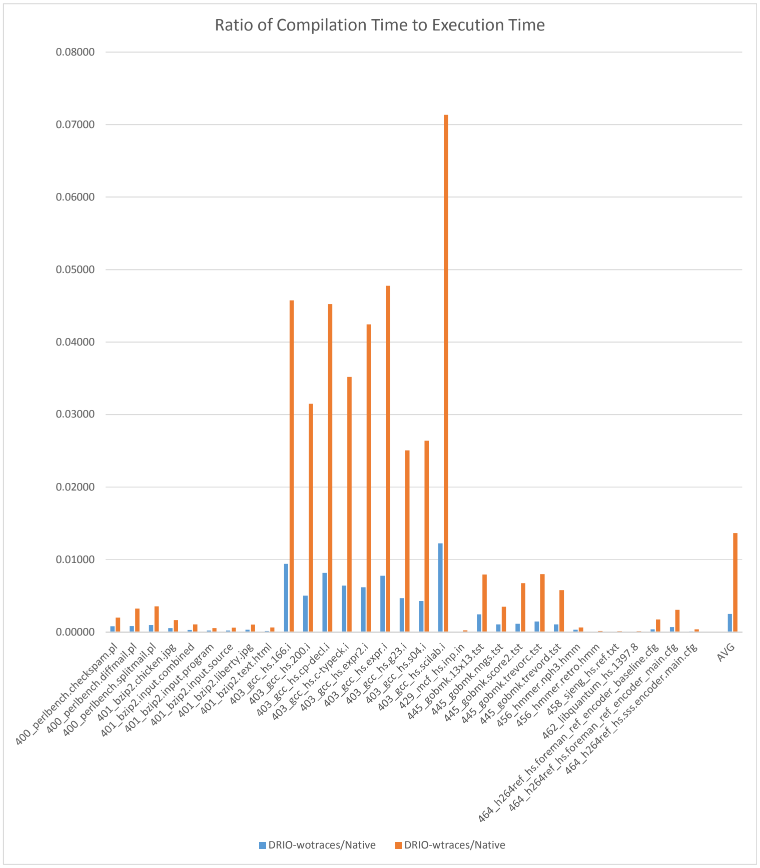

Figure 3.7. Ratio of *Compilation* time to *Execution* time for different benchmark ref inputs

Table 3.5 shows the application statistics for different benchmarks run with reference inputs like, trace counts, basic block counts, total context switches between the DynamoRIO and application contexts. The number of context switches are considerably more than the count for the fragments (traces, basic blocks) created during the execution, just like in the case of test inputs, leading to execution of high number of DynamoRIO instructions. But, because of the longer running programs or program inputs though the counts of context switches for ref inputs are similar to those of test inputs, the initial overhead of compilation (DynamoRIO

Benchmark	Trace Count	Basic Block Count	Native Run-Time(sec)	Total Context Switches
400_perlbench.checkspam.pl	8178.8	40578.0	227.6	541143.7
400_perlbench.diffmail.pl	4473.3	27557.7	71.7	340470.8
400_perlbench.splitmail.pl	6288.8	33740.1	106.0	450717.7
401_bzip2.chicken.jpg	734.0	4097.9	40.7	49749.7
401_bzip2.input.combined	749.0	4195.2	90.9	49779.3
401_bzip2.input.program	682.9	3960.8	126.4	47579.1
401_bzip2.input.source	763.9	4259.0	115.9	52278.9
401_bzip2.liberty.jpg	725.9	4006.9	62.9	47354.3
401_bzip2.text.html	684	3981.6	139.6	48431.5
403_gcc_hs.166.i	21874.2	108672.3	30.2	1462223.1
403_gcc_hs.200.i	25141.8	112271.7	46.5	1582352.4
403_gcc_hs.cp-decl.i	21800.1	103935.8	29.6	1416681.7
403_gcc_hs.c-typeck.i	22490.5	105935.2	43.6	1447296.1
403_gcc_hs.expr2.i	23836.8	110178.7	49.3	1529959.4
403_gcc_hs.expr.i	21163.6	102248.2	35.8	1413184.5
403_gcc_hs.g23.i	22641.7	106901.2	66.5	1468781.1
403_gcc_hs.s04.i	19306.1	96114.8	64.5	1283429.4
403_gcc_hs.scilab.i	24551.7	111303.3	17.4	1555177.4
429_mcf_hs.inp.in	260.0	2565.1	383.3	20240.5
445_gobmk.13x13.tst	9353.2	38286.0	70.8	575940.0
445_gobmk.nngs.tst	10011.8	40247.4	180.0	613075.2
445_gobmk.score2.tst	9214.0	37462.2	93.3	570369.9
445_gobmk.trevorc.tst	9423.4	38454.3	71.2	581499.0
445_gobmk.trevord.tst	9239.8	38129.1	96.7	571706.9
456_hmmer.nph3.hmm	748.2	6367.3	154.7	59207.8
456_hmmer.retro.hmm	409.0	4445.2	330.1	34410.4
458_sjeng_hs.ref.txt	1308.0	6272.3	594.0	83000.4
462_libquantum_hs.1397.8	240.0	2311.3	533.3	20141.5
464_h264ref_hs.foreman_ref_baseline.cfg	2268.0	12653.2	85.9	161563.5
464_h264ref_hs.foreman_ref_main.cfg	3042.0	15473.1	61.4	206221.5
464_h264ref_hs.sss_encoder_main.cfg	3109.0	15732.0	546.1	212284.6

Table 3.5. Different stats related to the benchmark ref inputs

code execution) could be compensated with the steady state execution leading to smaller percentages of compilation times as discussed with the Figure 3.7.

This means that the compilation is done less frequently compared to the test inputs. However, the overhead signifies extra instructions executed compared to native run. It would give us the two sources of DynamoRIO code and any penalty during execution of these additonal instructions. As discussed in the scenario of test inputs, the DynamoRIO instructions are executed when the control exits from the code cache and we have collected data for the reference inputs as well on the factors leading to fcache exits. The figure 3.8 shows the distribution of different factors causing the fcache exits for SPEC benchmarks with the ref input.

Even for reference inputs the outstanding factors causing the fcache exits are: *Indirect Branch targets and Link not allowed for trace heads.* Table 3.6 shows the fcache exit rate and the *perf* instuction count ratio for different benchmark reference inputs. The exit rates for the reference inputs is relatively less compared

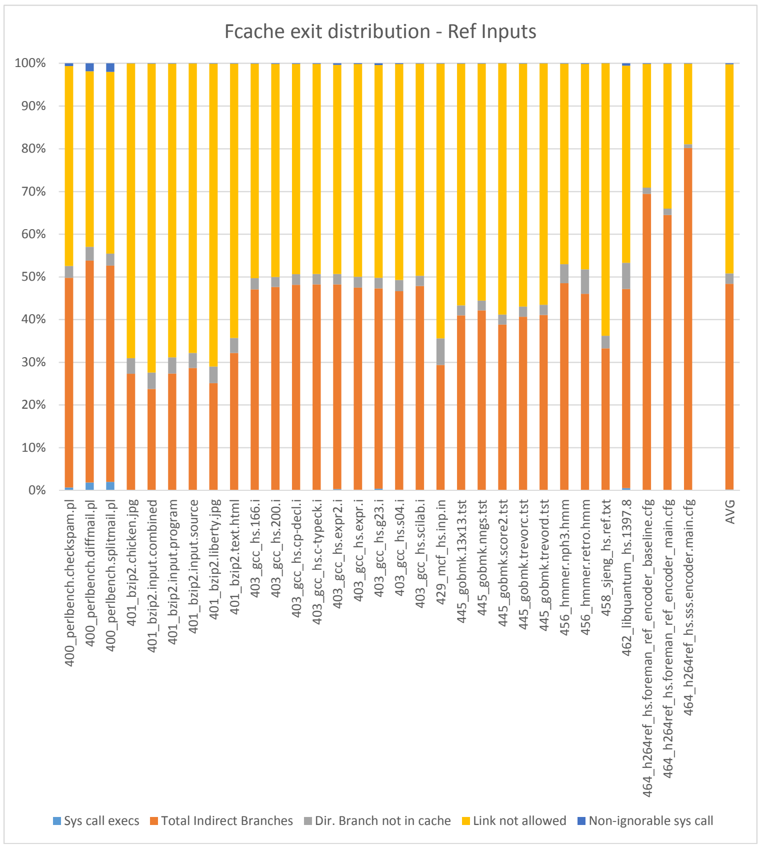

Figure 3.8. Distribution of different factors causing fcache exits with different benchmark reference inputs

to the test inputs, probably because of the longer steady state runs. Ideally, the high exit rate and instruction count ratio combination for *400.perlbench* should be high considering the high overheads in-order for our inference with test inputs to apply for reference inputs as well. However, the exit rates for *403.gcc* reference inputs are high but the *perf* instruction count ratio is highest for the *400.perlbench* benchmark reference input *diffmail.pl*. With the reference inputs it is not readily evident with the exit rate for the fcache exits to be the root cause of the overhead.

The instruction count distribution in different execution phases of reference inputs will shed some light on the evolving execution pattern as the steady state run progresses through completion. It would also show the influence of the op-

Benchmark	Total Fcache Exits/millisec	perf instrcnt ratio(DRIO/Native)
400_perlbench.checkspam.pl	1.35	1.902
400_perlbench.diffmail.pl	2.41	2.172
400_perlbench.splitmail.pl	2.5	1.815
401_bzip2.chicken.jpg	1.14	1.238
401_bzip2.input.combined	0.47	1.332
401_bzip2.input.program	0.32	1.297
401_bzip2.input.source	0.39	1.348
401_bzip2.liberty.jpg	0.71	1.261
401_bzip2.text.html	0.32	1.305
403_gcc_hs.166.i	34.34	1.933
403_gcc_hs.200.i	22.44	1.943
403_gcc_hs.cp decl.i	31.73	1.863
403_gcc_hs.c typeck.i	25.1	1.689
403_gcc_hs.expr2.i	24.04	1.684
403_gcc_hs.expr.i	29.95	1.74
403_gcc_hs.g23.i	17.55	1.721
403_gcc_hs.s04.i	15.57	1.722
403_gcc_hs.scilab.i	52.76	2.055
429_mcf_hs.inp.in	0.05	1.402
445_gobmk.13x13.tst	5.13	1.505
445_gobmk.nngs.tst	2.14	1.497
445_gobmk.score2.tst	4.36	1.425
445_gobmk.trevorc.tst	5.11	1.504
445_gobmk.trevord.tst	3.74	1.492
456_hmmer.nph3.hmm	0.41	1.057
456_hmmer.retro.hmm	0.11	1.076
458_sjeng_hs.ref.txt	0.1	1.669
462_libquantum_hs.1397.8	0.04	1.228
464_h264ref_hs.foreman_ref_baseline.cfg	2.72	1.527
464_h264ref_hs.foreman_ref_main.cfg	4.31	1.455
464_h264ref hs.sss_encoder_main.cfg	0.91	1.416

Table 3.6. *Fcache exit rates* and *perf* instruction count ratios for different benchmark reference inputs

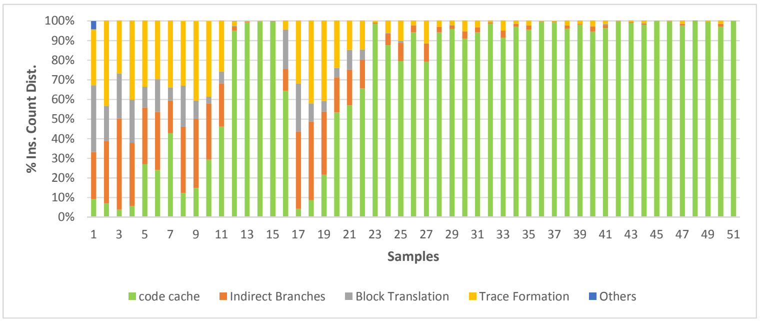

Figure 3.9. Instruction count distribution of benchmark 403.gcc for the input g23.i

timization techniques on the benchmark run, which was discussed with the test inputs as well. The Figure 3.9 shows the instruction count distribution in different execution phases of the benchmark *403.gcc* run with input *g23.i*. It can be seen that in the samples collected progressively with time, more instructions are being executed from within the code cache marked in color code *green*. This is the typical execution pattern expected for long running applications run through a binary translator. The optimization techniques incorporated in DynamoRIO [5] like, caching, linking, trace formation and instruction inlining have been able to considerably improve the execution time spent in the fragment cache. However, the additional instructions executed couldn't be fully alleviated through the optimization techniques as an overhead of around 30% could be seen in terms of run-time captured in the figure 3.6. The overhead would be accountable to the ability of the hardware to handle the excess instruction load. It is evident from the data given in the table 3.8 that the higher miss-rates of L1 icache, iTLB, branch-misses and context switches are higher than the native run, which couldn't be abated by the improved code locality in the fragment cache harvested during the steady state run.

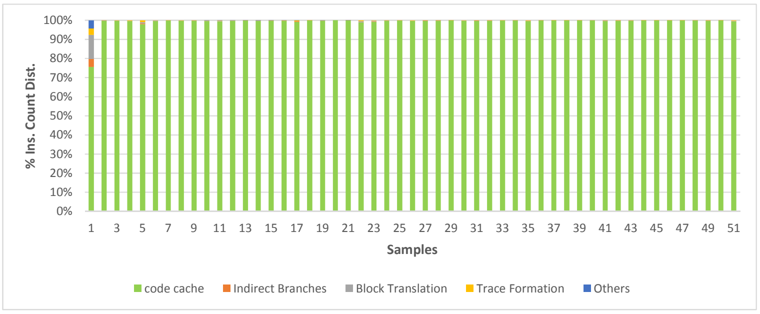

Figure 3.10. Instruction count distribution of benchmark 462.libquantum for the input parameters 1397 and 8

Another example is the instruction count distribution of the benchmark *462.libquantum* with input parameters *1397 and 8*, shown in the figure 3.10. Here, the execution reveals that a very high percentage of code has been executed from the fragment cache, and that the optimizations have been able to improve the performance compared to the native run, given the characteristic of longer run-time of the benchmark. This can be seen with the run-time stats portrayed in the figure 3.6. The ratio of DynamoRIO run-time to Native run-time for this benchmark input is little less than 1, indicating a speed-up.

Also, from the figure 3.8, it can be seen that for the benchmark *403.gcc* input *g23.i* and the benchmark *462.libqunatum* input parameters *1397 and 8*, the percentage of indirect branch exits from fragment cache is around 48%. However, from their respective instruction distributions shown in figure 3.9 and figure 3.10 respectively, not much of code has been executed in the corresponding execution phase for *Indirect Branches* with color code *orange*. This is an indication of the influence of instruction inlining to be able to resolve the indirect branches with the help of auxiliary code placed in fragment cache for the same.

Moving forward, we look into how the hardware has handled the extra instructions executed as a result of the fcache exits. As with the test inputs, the factors responsible for overhead with ref inputs also produce a cumulative effect with respect to the resulting overhead.

Table 3.7 shows the ratios of other code cache instruction counts to the total instructions, for different benchmark reference inputs. Similar to the test inputs the other instructions do not constitute much of the overhead. The highest is for the inputs of the benchmarks 458.sjeng and 462.libquantum with the overheads of 12 and 13 percent respectively. So, this is another source of an overhead, but not a significant one.

Benchmarks	Code Cache - Other ICount	Total ICount	CC-Other ICount/Total ICount
400_perlbench.checkspam.pl	20184472.000	1533744933.000	0.013
403_gcc_hs.g23.i	9631584.000	886058512.000	0.011
429_mcf_hs.inp.in	123841.000	41190975.000	0.003
445_gobmk.nngs.tst	813799.000	167865993.000	0.005
456_hmmer.retro.hmm	222048.000	62639472.000	0.004
458_sjeng_hs.ref.txt	11960064.000	99729147.000	0.120
462_libquantum_hs.1397.8	2970202773.000	22206986093.000	0.134
464.sss_encoder_main.cfg	144876.000	47774198.000	0.003

Table 3.7. Ratio of *Other code cache instructions* and *Total Instructions* of the benchmark reference inputs

Table 3.8 below shows the ratio's of *DynamoRIO* and *Application* hardware counters for different benchmark reference inputs.

Benchmark	Branch Miss%	Cache Miss%	Context Switches	L1 icache load misses	iTLB load misses
400_perlbench.checkspam.pl	0.87	0.57	2.72	3.60	17.24
400_perlbench.diffmail.pl	1.17	0.41	2.87	4.01	8.00
400_perlbench.splitmail.pl	1.24	0.75	3.27	4.71	7.50
401_bzip2.chicken.jpg	0.95	1.16	2.32	4.50	2.20
401_bzip2.input.combined	1.02	1.15	1.84	3.66	2.03
401_bzip2.input.program	1.02	1.10	2.05	3.24	2.00
401_bzip2.input.source	1.06	1.13	1.76	3.07	2.14
401_bzip2.liberty.jpg	0.85	1.32	2.23	3.79	2.35
401_bzip2.text.html	1.10	1.24	1.43	3.36	2.55
403_gcc_hs.166.i	1.31	1.23	2.22	7.04	15.16
403_gcc_hs.200.i	1.24	0.84	2.65	7.02	10.87
403_gcc_hs.cp-decl.i	1.63	1.66	2.68	7.58	12.16
403_gcc_hs.c-typeck.i	1.41	1.98	2.66	6.93	12.48
403_gcc_hs.expr2.i	1.27	1.75	1.71	6.90	9.58
403_gcc_hs.expr.i	1.43	1.73	1.33	7.49	8.73
403_gcc_hs.g23.i	1.08	1.74	1.69	6.68	9.46
403_gcc_hs.s04.i	1.46	0.72	1.21	6.28	12.50
403_gcc_hs.scilab.i	1.20	0.72	2.98	7.21	18.14
429_mcf_hs.inp.in	0.81	0.99	1.33	1.59	1.79
445_gobmk.13x13.tst	1.04	0.28	1.84	9.07	6.01
445_gobmk.nngs.tst	1.07	0.30	1.88	9.50	8.27
445_gobmk.score2.tst	0.99	0.11	1.99	9.02	6.21
445_gobmk.trevorc.tst	1.04	0.22	1.64	9.53	6.22
445_gobmk.trevord.tst	1.03	0.22	1.89	10.03	6.42
456_hmmer.nph3.hmm	1.03	1.12	1.37	2.77	0.80
456_hmmer.retro.hmm	0.91	1.06	1.30	1.89	0.82
458_sjeng_hs.ref.txt	0.84	0.60	2.55	56.01	4.00
462_libquantum_hs.1397.8	0.77	0.92	1.70	1.40	2.03
464_h264ref_hs.foreman_ref_baseline.cfg	0.99	0.99	2.35	5.87	2.84
464_h264ref_hs.foreman_ref_main.cfg	1.03	0.85	2.42	6.51	2.74
464_h264ref_hs.sss_encoder_main.cfg	0.94	0.90	1.83	5.08	2.63

Table 3.8. Ratio of *DynamoRIO* and *Application PERF* counters for different events with each of the benchmark ref inputs

As stated earlier, the overhead is the result of the cumulative effect of different factors rather than a single stand-out factor. Like with the *perf* stats, for different benchmarks it can be seen that during the execution, though few factors result in a speed-up like improved overall *branch prediction* and *cache miss percentage*, other factors like *Context Switches*, *L1 icache load misses* and *iTLB load misses*,result in a slow-down nullifying the affect of the speed-up gained with the earlier factors. The two factors: *L1 icache load misses* and *iTLB load misses* have the dominating effect on the overhead. The inputs for the larger benchmarks *400.perlbench*, *403.gcc* and *445.gobmk* have comparatively higher misses resulting in higher overheads. All the benchmarks with overheads close to and above 50% show a combination of these factors irrespective of the size of the benchmarks. To explain the highest overhead with the *400.perlbench* inputs, we should look at the absolute hardware counter data of all the benchmarks, as the numbers in table 3.8 are for each of the benchmarks with respect to native run and not relative to other benchmarks.

The table 3.9 shows the absolute hardware counter values for all the benchmarks. The *400.perlbench* inputs have a moderate number for each of the factors resulting in a higher overhead. Other benchmarks do have higher numbers than the perlbench but other factors were not has high or not that moderate. The traces did help the reference inputs with an improved code locality which can be seen with relatively improved cache misses compared to native run. But in-order to execute the DynamoRIO instructions for the caching, the penalty outweighed the resultant benefits.

Benchmark	page faults	context switches	L1 icache load misses	iTLB load misses
400_perlbench.checkspam.pl	62719.0	2086.2	29394939703.8	461132463.6
400_perlbench.diffmail.pl	60701.4	752.0	15968361691.4	108681967.1
400_perlbench.splitmail.pl	177615.2	1191.2	6456686207.2	31362344.1
401_bzip2.chicken.jpg	19010.0	244.6	35705777.3	572765.4
401_bzip2.input.combined	106419.0	480.9	65677168.8	1432250.7
401_bzip2.input.program	145827.0	538.6	85777371.6	1985356.0
401_bzip2.input.source	144809.1	439.3	76222039.8	1913225.1
401_bzip2.liberty.jpg	19332.5	340.9	42586662.9	859808.2
401_bzip2.text.html	147179.3	646.6	93104456.4	2571532.2
403_gcc_hs.166.i	145114.1	255.5	2401657380.2	19724103.6
403_gcc_hs.200.i	100230.6	376.7	6294751868.9	40345460.6
403_gcc_hs.cp decl.i	132964.6	304.2	2308228664.3	18040525.6
403_gcc_hs.c typeck.i	204064.1	368.3	3260020837.3	25738947.0
403_gcc_hs.expr2.i	359328.2	452.3	2798556907.7	22925898.6
403_gcc_hs.expr.i	229253.8	396.1	2008010664.2	15843322.2
403_gcc_hs.g23.i	318081.3	586.3	2640536811.3	23782506.3
403_gcc_hs.s04.i	437967.3	647.1	2923525056.8	31539677.1
403_gcc_hs.scilab.i	42897.3	206.8	3489980001.0	34095804.8
429_mcf_hs.inp.in	319262.6	1026.5	115111931.1	2943680.8
445_gobmk.13x13.tst	7659.1	570.7	24903682383.4	22517451.0
445_gobmk.nngs.tst	7988.1	1294.8	63367259168.2	71465280.1
445_gobmk.score2.tst	7623.0	619.9	20851630193.4	24926334.7
445_gobmk.trevorc.tst	7769.6	675.4	26939805062.7	21952245.6
445_gobmk.trevord.tst	7699.7	743.3	33434956140.6	28851708.1
456_hmmer.nph3.hmm	7810.7	506.7	273819502.8	2300921.7
456_hmmer.retro.hmm	1962.2	887.0	182427750.6	3099690.7
458_sjeng_hs.ref.txt	29241.6	2625.0	126710850939.4	69682253.5
462_libquantum_hs.1397.8	145412.5	1385.8	175130571.5	3693486.8
464_h264ref_hs.foreman_ref_baseline.cfg	10701.6	534.9	1125155452.3	6115595.7
464_h264ref_hs.foreman_ref_main.cfg	7367.3	376.5	1587285881.0	5880242.1
464_h264ref hs.sss_encoder_main.cfg	22109.8	2656.2	10664863334.9	45536380.1

Table 3.9. *DynamoRIO* hardware numbers for reference inputs

In summary, not just the fcache exit rate, but the exit rate combined with the impact on hardware can result in an overhead or speed up. To address the issue of overhead, research is to be done on how to reduce the number of fcache exits because of the two major factors: *Indirect Branch Target and Linking not allowed with trace head* and also in the design of an efficient hardware to handle the high instruction volume. This can include larger TLB tables, page tables and more history to be accounted for by the branch predictor as both the DynamoRIO code and application code are being run from within a single process without any threads resulting in high pressure on the hardware. Options for parallelization within the DynamoRIO code base and eager translation to compile basic blocks ahead of time(analogous to pre-fetching) can be explored to run things faster.

Chapter 4

Related Work

Prior research has been done to explore different reasons for the overhead of running applications under Dynamic Binary Translation Systems. Techniques to address different causes of the overheads have been researched, implemented and shown to improve run-times. The work of Arkaitz Ruiz and Kim Hazelwood [27] in exploring how the hardware is affected by running applications from within the DBT's is close to our work. In their work they have used the *perf* [25] tool to get the hardware counters for different hardware events during the execution. They use *Pin* [4] and *DynamoRIO* [12] for their experiements. Their focus was mainly in exploring the impact of *Pin* on the hardware and the causes. They have run the SPEC2006 INT benchmarks [9] under both *Pin* and *DynamoRIO*. Acccording to their results, the main root cause of the overhead is the high volume of instructions executed than the native execution, causing high L1 instruction cache misses and iTLB misses for most of the benchmarks. We have adopted the *perf* tool as well to collect our hardware statistics during benchmark execution and our results validate their observations.

We not only explore the hardware impact but other aspects of the DBT as well, like the number of exits, basic block's and traces created during execution, run-time and compilation-time during the application execution inorder to achieve a more wholesome understanding of how the DBT influences the application execution. Certainly, the root cause of the overhead is high instruction execution derived from the large code base of DynamoRIO executed during the translation. This is the result of the control exiting the code cache for the translation. In our work we found that the main reasons for the exits from the code cache is indirect branch execution and no linking between basic blocks and trace heads, so, the ultimate cause of overhead filters down to the source pc to target pc translation in executing basic blocks and traces from within the respective code caches.

Research has been done in better handling of indirect branches previously and SPIRE [22] is an approach to handle hot indirect branches during source pc translation. The idea is not to exit the code cache for the translation, but to have a trampoline at the source pc address that would in turn redirect the control to the code cache for executing the translated code. There can be self-modifying applications and as SPIRE system would modify the original source code with a trampoline, there is a need to maintain code transparency to the application. For this a code space with size of original source binary is created and the trampolines are created in the new space, leaving the original source code untouched. With the hot indirect branches handled by SPIRE, the overhead of context switching for translation can be reduced to improve the performance.

The method used in SPIRE is probe-based and DynamoRIO uses JIT-based compilation with *software prediction* as the method to resolve the indirect branches. With *software prediction* a compare-jump list is put in place for the indirect branch to jump to the appropriate target pc, as the source pc can't be known until the branch instruction had been executed. With the compare-jump list, only the corresponding mappings of source pc's in the list have to be checked from the hash-table with all the source to target pc mappings. However, if the mapping can't be found from within the list, the whole lookup-table has to be checked for resolving the indirect branch. In the prior work with SPIRE, it has been shown on indirect branch intensive benchmarks that the probe-based method in SPIRE is more efficient than the *software based* approach currently implemented in DynamoRIO.

Apart from SPIRE approach to reduce the instances of exiting code cache for address translation, the work on HQEMU [20] has also shown to help in reducing the exits when executing traces from within code cache. With HQEMU, the goal is to improve the quality of translated code with additional compilation of intermediate code from QEMU with LLVM translation. LLVM is an optimization intensive compiler. In addition to improved code quality a technique called *trace merging* has been implemented with HQEMU to address the exits while executing the traces from code cache. When traces are created and executed, it is possible that the control is switched between certain traces more frequently. With *trace merging*, depending on information provided by hardware called *Hardware Performance Monitor* data on such traces is collected and are merged into a single trace avoiding the switching between the code cache and DBT for the trace address translation. These would be hot traces that would be executed more frequently and therefore a large number of exits from code cache can be averted, therefore reducing a considerable amount of overhead due to translation.

So, these techniques of reducing the translation overhead with the SPIRE system and *trace merging* are of interest to address the two major reasons for code cache exits: *Indirect Branches and No linking between fragments in code cache with trace heads*.

Chapter 5

Future Work

Our work in this thesis studies the major causes for performance overhead in DBT compared to native program execution. Based on our results and observations, there are several avenues for future work.

One prominent cause of the performance overhead is the large number of exits from the code cache that are required to service various DBT tasks. Our immediate focus in the future will be to explore techniques to reduce the number of exits from the code cache. We intend to develop and evaluate different mechanisms to reduce the number of code cache exits for each of the three main causes: basic block translation, indirect branches, and trace formation. We expect our techniques to use the additional computation resources of multi-core and many-core machines to parallelize these tasks and allow the main DBT program execution to stay in the code cache for longer.

More work is also need to resolve the impact on hardware caused by high instruction volume and loss of instruction (and perhaps, data) locality due to DBT execution. We will research and design better hardware techniques to efficiently execute DBTs with larger code bases. We will also attempt to combine our software and hardware techniques to create a more collaborative effort to achieve an ideal DBT execution environment that can run applications close to or even better than native execution performance on modern multi-core machines.

Chapter 6

Conclusions

Dynamic binary translation is an important mechanism to realize portable program execution, program profiling for performance improvement, and monitoring and instrumentation to provide a secure execution environment. Unfortunately, program execution in a DBT can cause small to significant performance overhead, resulting in limiting the adoption of this technology in mainstream systems. Our goal in this thesis was to understand the causes for performance overhead in a DBT to allow targeted resolution of such causes in the future.

We have designed new experiments to explore performance characteristics for DynamoRio, which is a popular and sophisticated x86 to x86 binary translator, instrumentor and optimizer. The performance overhead of a DBT is due to a combination of factors. Our experiments measuring the effect of program execution in a DBT on processor cache and branch prediction hardware reveal trends that are mostly consistent with earlier results. In particular, we confirmed that program execution in a DBT exerts greater pressure on the L1 instruction cache and the instruction TLB due to a higher volume of executed instructions compared to native program execution.

In this thesis we conducted further experiments to understand the causes for the higher volume of executed instructions and associated performance overheads. We found that the performance overhead can be attributed to frequent exits from the code cache to service the additional tasks performed by the DBT. We determined that the major causes for code cache exits include *block translation*, *trace formation*, and *indirect branch resolution*. Our experiments and graphs also reveal the number of exits in each category and the number of instructions executed to service such events throughout program execution. Based on these results we plan to devise parallelization techniques to conduct these services asynchronously and in advance to reduce the number of code cache exits and DBT performance overhead in the future.

Appendix A

Instruction Count Distribution Graphs - Test Inputs

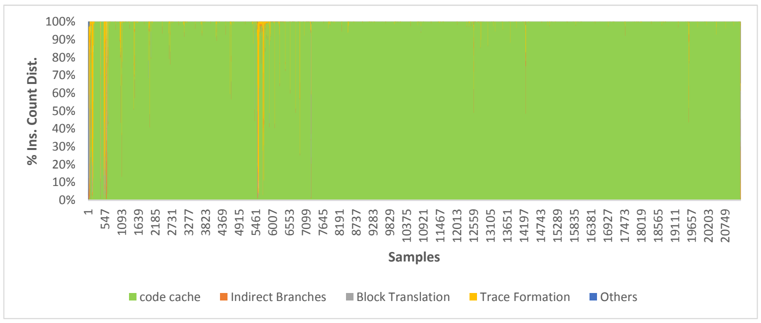

Figure A.1. Distribution of instruction counts for different execution phases of 401.bzip2 benchmark test input input.program

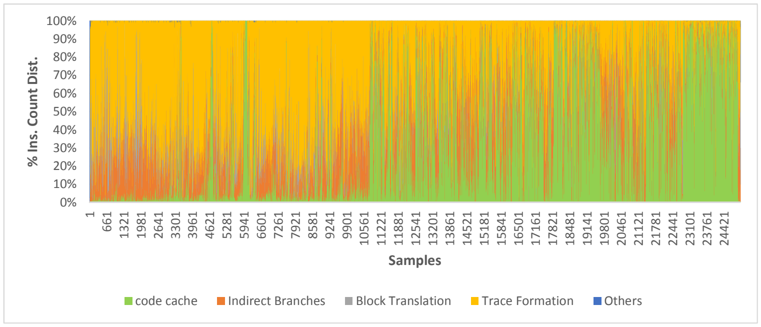

Figure A.2. Distribution of instruction counts for different execution phases of 403.gcc benchmark test input cccp

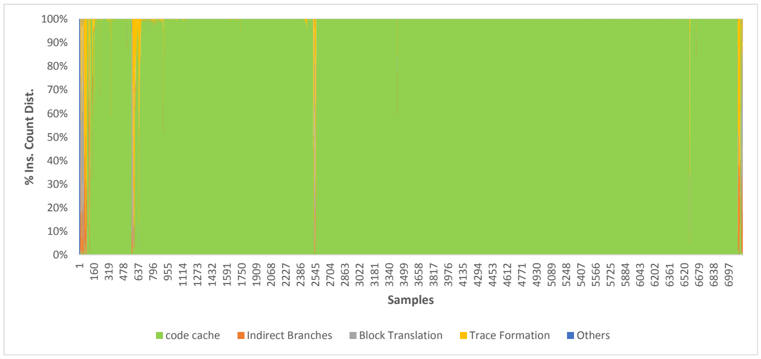

Figure A.3. Distribution of instruction counts for different execution phases of 429.mcf benchmark test input inp.in

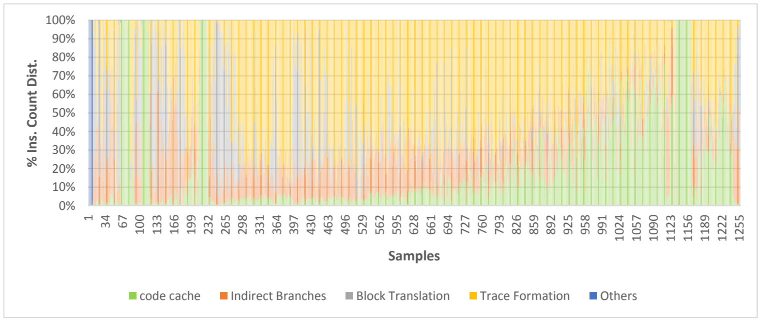

Figure A.4. Distribution of instruction counts for different execution phases of 445.gobmk benchmark test input connect.rot

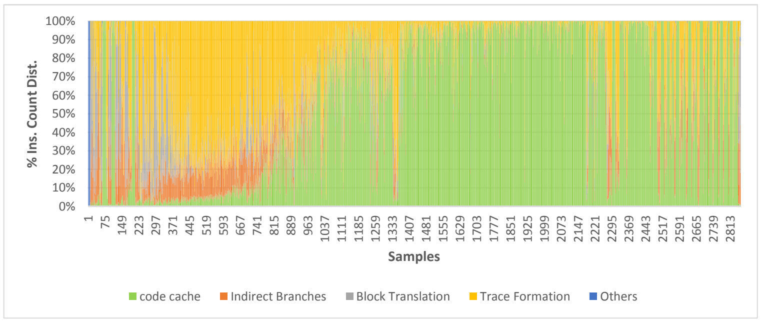

Figure A.5. Distribution of instruction counts for different execution phases of 445.gobmk benchmark test input cutstone

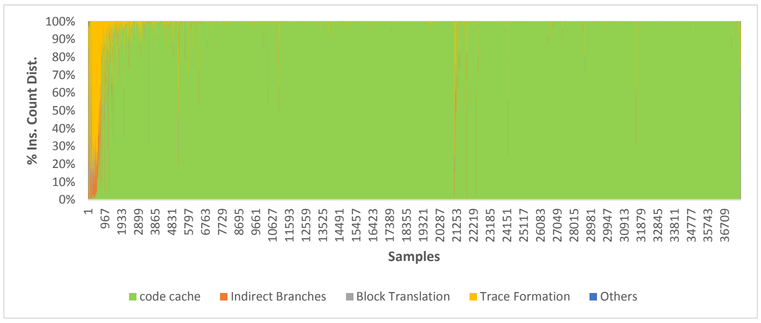

Figure A.6. Distribution of instruction counts for different execution phases of 458.sjeng benchmark test input test.txt

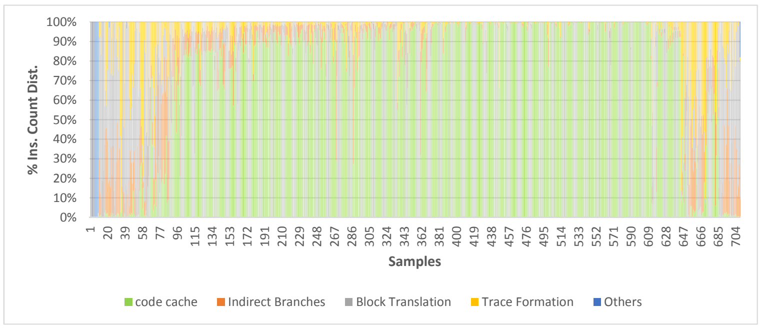

Figure A.7. Distribution of instruction counts for different execution phases of 462.libquantum benchmark test input parameters 33 and 5

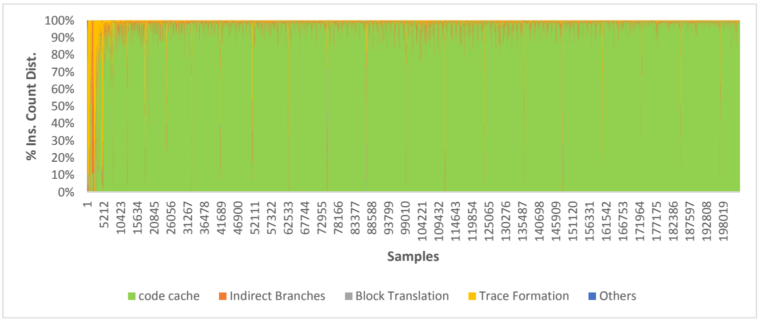

Figure A.8. Distribution of instruction counts for different execution phases of 464.h264ref benchmark test input foreman

Appendix B

Instruction Count Distribution Graphs - Reference Inputs

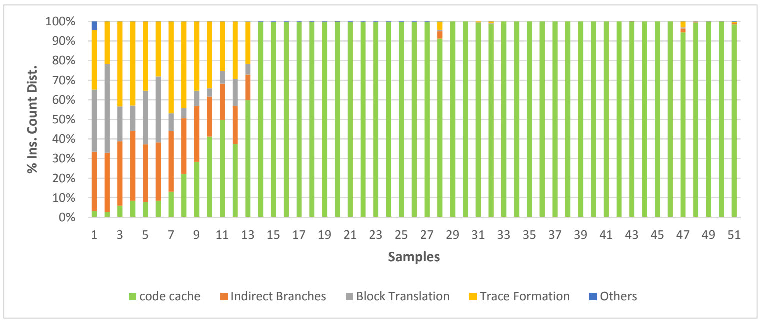

Figure B.1. Distribution of instruction counts for different execution phases of 400.perlbench benchmark ref input checkspam.pl

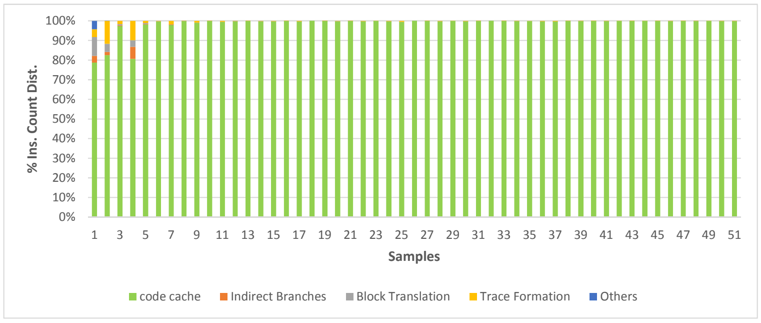

Figure B.2. Distribution of instruction counts for different execution phases of 401.bzip2 benchmark ref input.program

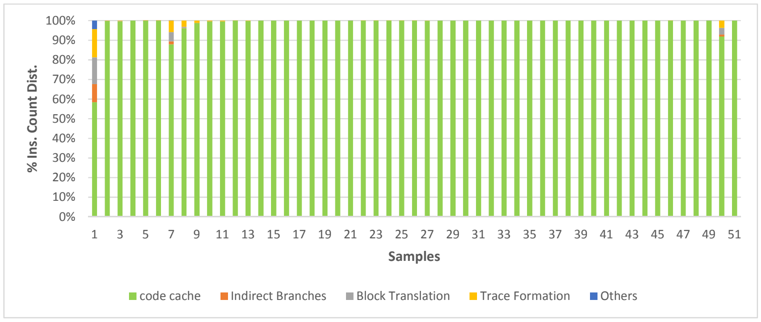

Figure B.3. Distribution of instruction counts for different execution phases of 429.mcf benchmark ref input inp.in

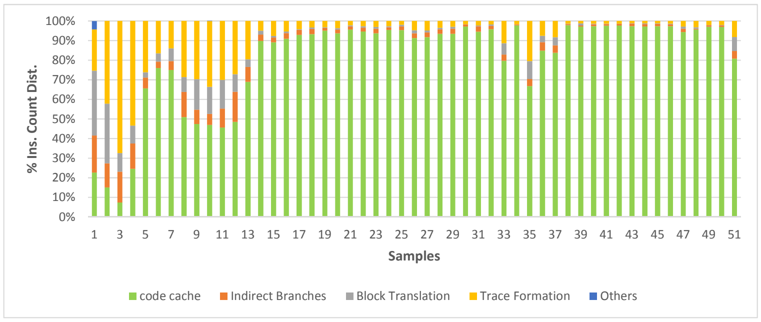

Figure B.4. Distribution of instruction counts for different execution phases of 445.gobmk benchmark ref input nngs

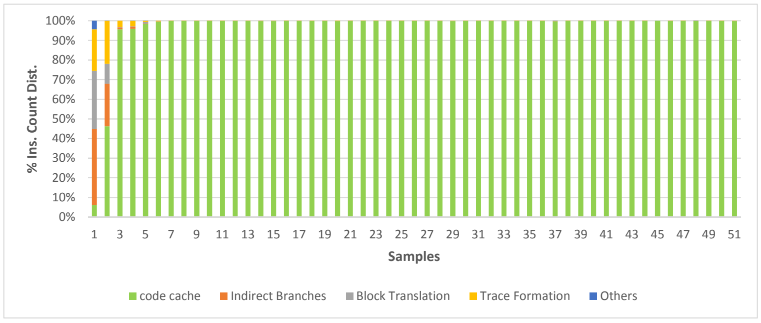

Figure B.5. Distribution of instruction counts for different execution phases of 456.hmmer benchmark ref input retro.hmm

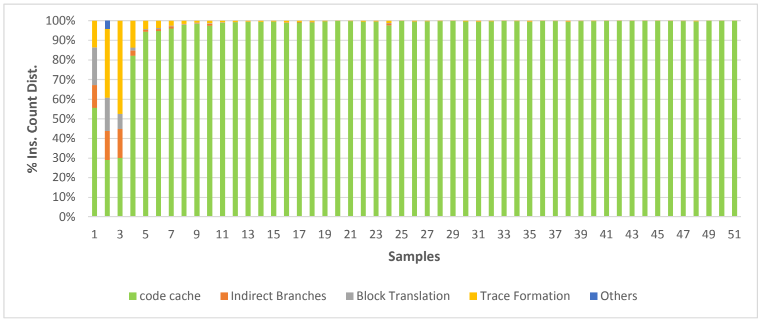

Figure B.6. Distribution of instruction counts for different execution phases of 458.sjeng benchmark ref input ref.txt

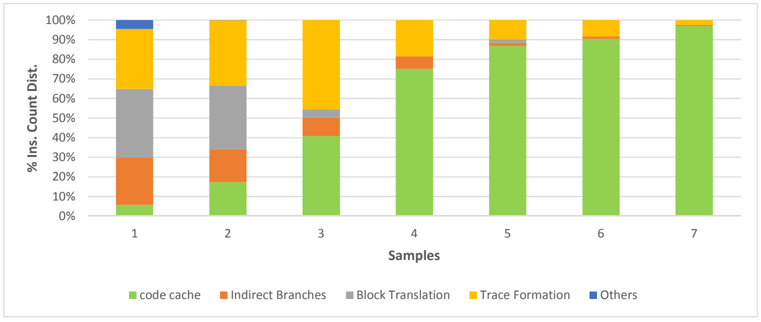

Figure B.7. Distribution of instruction counts for different execution phases of 464.h264ref benchmark ref input sss.encoder.main.cfg

References

[1] About.com. nm-linux command-unix command. http://linux.about.com/library/cmd/blcmdl1_nm.htm, May 2015.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] S. Berkowits. Pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, June 2012.

[5] D. L. Bruening. Effcient,transparent,and comprehensive runtime code manipulation. http://www.burningcutlery.com/derek/docs/phd.pdf, September 2004.

[6] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004. AAI0807735.

[7] R. F. Cmelik and D. Keppel. Shade: A fast instruction set simulator for execution profiling. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1993.

[8] W.-T. Command. time - unix command. http://en.wikipedia.org/wiki/Time_%28Unix%29, October 2014.

[9] S. S. P. E. Corporation. Cint2006 - integer component of spec cpu2006. https://www.spec.org/cpu2006/CINT2006/, August 2006.

[10] Q. Z. Derek L. Bruening. Deployment - linux platform. http://dynamorio.org/docs/page_deploy.html, September 2014.

[11] Q. Z. Derek L. Bruening. Dynamorio - building a client. http://dynamorio.org/docs/using.html#sec_build, September 2014.

[12] Q. Z. Derek L. Bruening. Dynamorio - dynamic instrumentation tool platform. http://dynamorio.org/, September 2014.

[13] Q. Z. Derek L. Bruening. Dynamorio: Common events. http://dynamorio.org/docs/using.html#sec_events, September 2014.

[14] Q. Z. Derek L. Bruening. Fine-tuning dynamorio: Runtime parameters. http://dynamorio.org/docs/using.html#sec_options, September 2014.

[15] die.net. ptrace(2)-linux man page. http://linux.die.net/man/2/ptrace, February 2015.

[16] DynamoRIO. Dynamorio api. http://dynamorio.org/docs/index.html, September 2014.

[17] DynamoRIO. Dynamorio system details. http://dynamorio.org/docs/overview.html, September 2014.

[18] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.

[19] J. D. Hiser, D. W. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. *ACM Trans. Archit. Code Optim.*, 8(2):9:1–9:28, June 2011.

[20] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104–113, New York, NY, USA, 2012. ACM.

[21] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 2–12, New York, NY, USA, 2006. ACM.

[22] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang. Spire: Improving dynamic binary translation through spc-indexed indirect branch redirecting. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 1–12, New York, NY, USA, 2013. ACM.

[23] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[25] PerfWiki. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial, May 2014.

[26] A. Ruiz-Alvarez and K. Hazelwood. Evaluating the impact of dynamic binary translation systems on hardware cache performance. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 131–140, Sept 2008.

[27] A. Ruiz-Alvarez and K. Hazelwood. Evaluating the impact of dynamic binary translation systems on hardware cache performance. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 131–140, Sept 2008.

[28] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. Technical report, Charlottesville, VA, USA, 2001.

[29] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[30] tutorialspoint.com. Fork-system call. http://www.tutorialspoint.com/unix_system_calls/fork.htm, May 2015.

[31] D. Ung and C. Cifuentes. Dynamic binary translation using run-time feedbacks. *Sci. Comput. Program.*, 60(2):189–204, Apr. 2006.

[32] wikipedia.org. Kill-unix command. http://en.wikipedia.org/wiki/Kill_%28command%29, May 2015.

[33] wikipedia.org. Sleep-unix command. http://en.wikipedia.org/wiki/Sleep_%28Unix%29, May 2015.